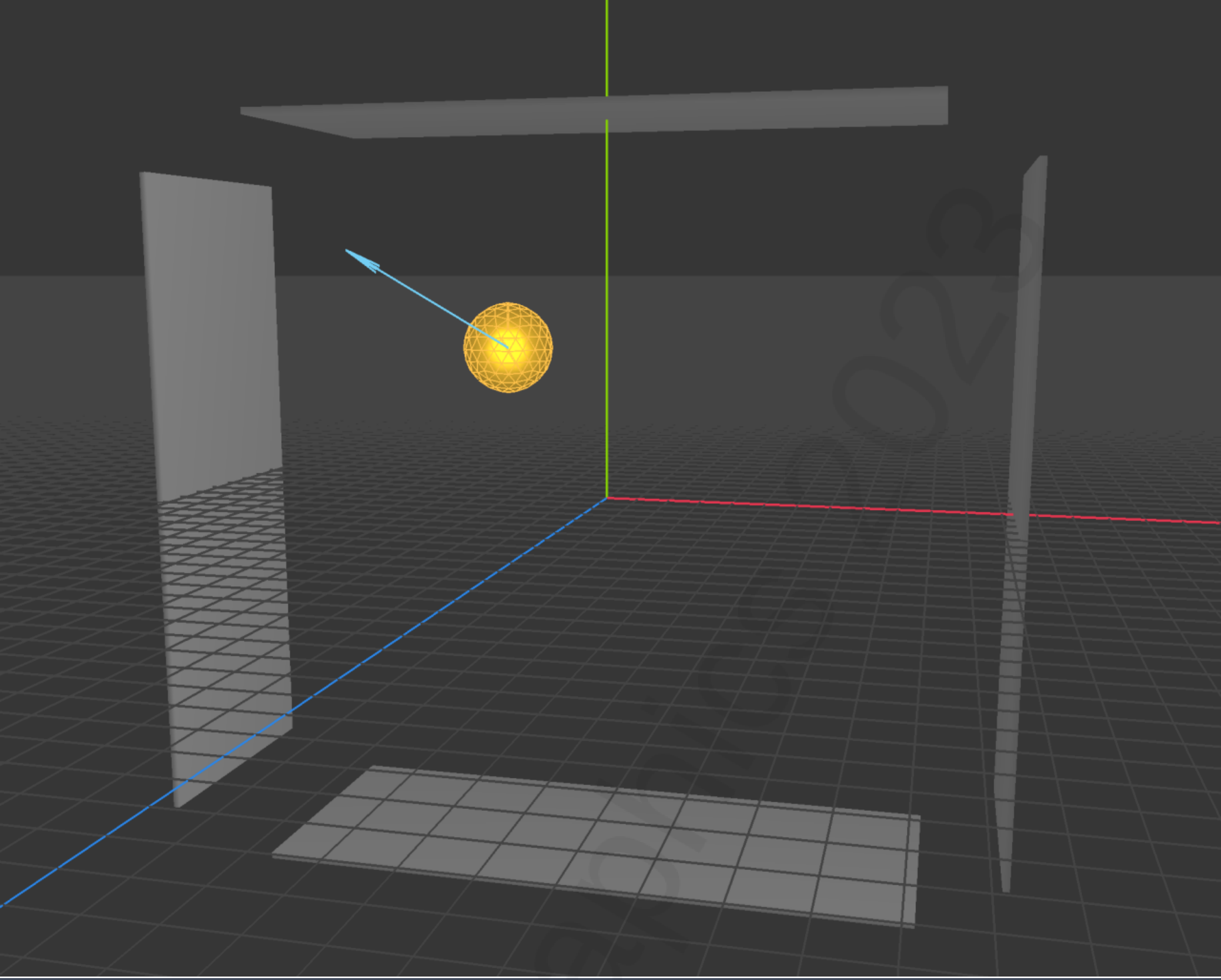

西安交通大学计算机图形学实验文档

物理模拟部分

作者：罗思源 李昊东 闫青云

组织：计算机图形学课题组

时间：September 18, 2023

目录

序

这部分文档是计算机图形学物理模拟部分选做实验的介绍。在这部分实验中，你将模拟物体的运动规律，并以此生成动画效果。这需要你对运动学有基本的了解，并阅读关于碰撞检测的资料。

这部分文档总共包含四个选做实验，分别是：

- 前向欧拉法模拟运动：用最简单的方法模拟物体的平动。
- 更好地求解运动方程：改进的运动求解方法。
- 朴素的碰撞检测：检测和处理物体之间相互碰撞的过程。
- 用几何数据结构加速碰撞检测：使用 BVH 加速碰撞检测的过程。

在做实验之前，你可能需要复习一点大学物理和常微分方程的知识，不过我们并不涉及很深的物理和数学知识，总体上还是比较简单的。[1][2]

[1]这份文档使用 ElegantBook 模板编写，按 CC BY-NC-SA 4.0 协议发布。

[2]封面图来自一颗小球在四面挡板之间来回碰撞的动画，我们截取了其中一帧。

第 1 章 前向欧拉法模拟运动

质点运动是最简单的运动形式，使用前向欧拉法求解运动方程、模拟质点运动（实验 2.12），就可以让物体运动起来。

1.1 实验内容

回顾质点在恒定外力作用下的匀变速运动过程，利用前向欧拉法求相应运动方程的数值解并更新物体的运动状态，从而产生动画效果。

1.2 指导与要求

1.2.1 物理模拟与动画

如果渲染一系列内容变化较为平缓的图像，那么连续播放这些图像就会产生动画效果。很多旧式动画是由美术工作者手工绘制的，艺术家根据自己对真实世界中运动过程的了解，将想象中的物体运动过程分解成“帧”画在纸上。在真实世界中，物体的运动过程遵循各种物理规律。而动画画面上的运动过程至少看起来要遵循相同的规律，才不会让观众感到难受。这对艺术家提出了很高的要求，只有长久的学习和训练才能准确地“想象”一个物体是如何运动的。

在图形学的思维框架下，我们首先用模型记录物体的三维形状，然后用渲染的方法将三维形状转换成二维的图像。在渲染时，这种过程保证不同视角下的渲染结果具有“形状一致性”：无论我们看到哪个角度的渲染图，都会认为图中的物体形状是相同的。

一般的运动也是在三维空间中进行。我们可以类比“形状一致性”去建立“运动一致性”：首先在三维空间中不断计算物体运动的数据并更新模型的参数，然后将变化后的场景渲染成新的图像，那么我们也可以得到一系列描述运动过程的图像。由于我们直接在三维空间下模拟物体的运动，模型位置（或形状）的变化过程自然会符合真实的运动规律，渲染出的图像序列看起来就会很合理，且合理性是由三维空间中的物理规律本身保证的。这种替代人工生成真实感动画的方法，就是**物理动画**，即使用物理模拟的方式生成动画。

1.2.2 最简单的物理动画：物体的自由运动

在经典物理学框架下，最简单的运动模型莫过于质点运动模型。在这个模型中，一个物体的大小和形状都被忽略，所有质量集中于一点，这一点具有位置、速度、加速度等运动学属性。如果场景中的物体之间不产生交互，我们又不考虑物体的转动，那么质点运动模型已经可以生成一段比较合理的动画了。图 1.1 就是一个简单的例子。

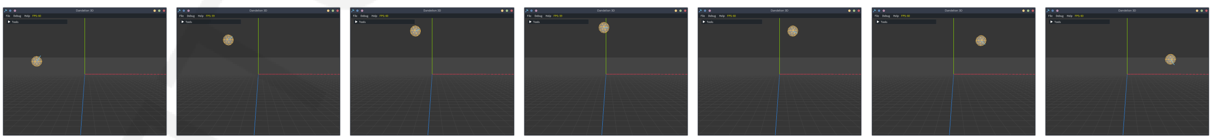

图 1.1: 按照质点运动学模拟一个小球的抛体运动过程（仅作示意，截图的时间间隔并不相等）

在质点运动模型中，质点的运动可以用两个微分方程描述：

$$
\begin{aligned}
\mathbf{v} &= \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} \\
\mathbf{a} &= \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}
\end{aligned}
\tag{1.1}
$$

取加速度 $\mathbf{a}$ 为一常数，再给定一个初始条件（边界条件）$\mathbf{x}(0) = \mathbf{x}_0$ ，方程组 1.1 就有唯一解。解出的函数 $\mathbf{x}(t)$ 就代表质点（物体）的运动轨迹。

你当然可以手算出方程组 1.1 的解析解。但只要运动过程稍微复杂一些（比如加速度不定，或者发生碰撞），解析解的形式就会相当复杂，甚至很难求出解析解。相应的解决方案就是微分方程数值解法，即在不求出函数表达式的情况下直接求出若干采样点处的函数值。通常我们使用等距采样点，用 $\mathbf{x}_k$ 表示经过了 $k\Delta t$ 时间后物体的位置。

由于微分是差分的极限形式，用差分逼近微分是一件很自然的事情。将方程组 1.1 改写成差分形式就是方程组 1.2 。

$$
\begin{aligned}
\mathbf{v} &= \frac{\mathbf{x}_{k+1} - \mathbf{x}_k}{\Delta t} \\
\mathbf{a} &= \frac{\mathbf{v}_{k+1} - \mathbf{v}_k}{\Delta t}
\end{aligned}
\tag{1.2}
$$

差分方程中不含极限运算，两边乘上分母 Δt 就可以得到一个递推方程：

$$
\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{v}\Delta t \\
\mathbf{v}_{k+1} &= \mathbf{v}_k + \mathbf{a}\Delta t
\end{aligned}
$$

指定好 $\mathbf{x}_0, \mathbf{v}_0, \mathbf{a}$ 这三个参数后，就可以逐步迭代求解位置和速度了。

1.2.3 渲染帧率和模拟帧率

我们模拟运动的目的是生成动画，因此最终还是要将运动结果渲染成图像。如果用离线渲染的方法渲染动画，那么每过 Δt 渲染一帧图像就可以，这时渲染帧率是固定的，且与模拟帧率相同。然而如果我们希望看到实时动画，就必须解决两个关于渲染帧率的问题：

- 实时渲染的帧率是不固定的，相邻两帧画面的时间间隔不一定等长。
- 像 OpenGL 这样的图形 API 是通过调用特定的 API 交换前后缓冲来显示下一帧画面的，我们通常不会定时调用（就是所谓的“锁帧率”），而是每次执行完计算过程后立即调用（也就是尽可能维持高帧率）。因此，渲染帧率和模拟帧率之间没有任何固定关系。

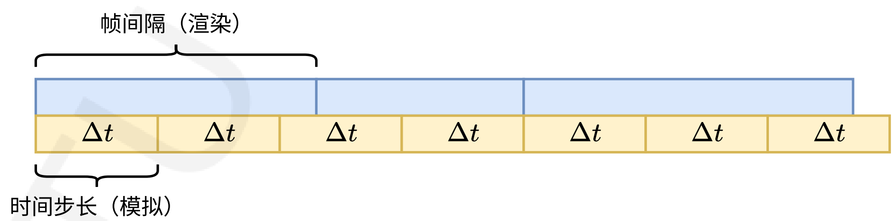

图 1.2: 实时渲染帧间隔与模拟时间步的关系

图 1.2 中展示了实时渲染过程中帧间隔与模拟步长之间的关系，应该有助于你理解刚才提到的两个问题。为了更方便地求解差分方程，我们希望模拟过程的步长 Δt 是固定不变的；而实时渲染的帧间隔是变化的，因此我们渲染每一帧前都要决定模拟多少个时间步。

我们称现在画面上的图像为“当前帧”，在当前帧之前的是上一帧，即将被渲染到屏幕上的是下一帧。Dandelion 的解决方法是：在当前帧渲染完成后，取帧间隔作为剩余时长，循环消耗这个时长直到它不足一个 Δt，模拟完

成后渲染下一帧。下面的伪代码展示了模拟的过程，请注意其中的帧间隔定义与图 1.2 略有不同（`last_update` 一定是某个模拟时间步的结尾）。

```
for each object
    initialize x, v and a
last_update = t_0

while simulating
    frame_duration = t_now - last_update
    remained_duration = frame_duration
    while remained_duration > time_step
        for each object
            update each its state
        remained_duration = remained_duration - time_step
    last_update = last_update + simulation_duration_within_this_frame
    call swap_buffer() to render a new frame
```

1.2.4 实现细节

你可以用 C++ 标准库提供的 `time_point` 和 `duration` 两种类型存储时间点和时间段，用 `steady_clock::now()` 获取当前时间。请自己查阅以下文档了解如何实用这些时间类型：

- std::chrono::time_point
- std::chrono::duration
- std::chrono::steady_clock

在你需要记录和计算时间的文件中，我们已经作了如下的定义：

```
using std::chrono::steady_clock;
using time_point = std::chrono::time_point<std::chrono::steady_clock>;
using duration   = std::chrono::duration<float>;
```

因此你无需关心 `time_point` 和 `duration` 的各种模板参数，只需要直接使用 `time_point` 和 `duration` 就可以了。在我们的实验中，时间都是以秒为单位计算的。

模拟的时间步长（即 Δt ）是 `time_step` ，你必须使用这个变量存储的时间步长，而不能随意写一个常数。

在 Dandelion 中，物体的物理属性包括位置、速度、合外力和质量四种，它们都可以在图形界面上设置。另外，我们定义了 `KineticState` 类型表示物体的运动状态，请阅读开发者文档：KineticState 结构体来了解它的属性。

1.2.5 要求

你需要完成的函数有：

- *scene/scene.cpp* 中的 `Scene::simulation_update` 函数，在其中实现计算帧间隔和执行模拟的过程。

- *scene/object.cpp* 中的 `Object::update` 函数，在其中根据模拟结果更新物体的运动学状态。这个函数也负责碰撞检测，但本次实验中直接忽视这部分代码即可。
- *simulation/solver.cpp* 中的 `forward_euler_step` 函数，在其中实现用前向欧拉法迭代一步的过程。

完成这些函数后，请加载 *cube.obj* 。首先在布局模式下将方块的位置设为 $(-5, 0, 0)$ ，然后在物理模拟模式下将初速度设置为 $(2, 3, 0)$、合外力设置为 $(0, -1, 0)$ ，点击 *Start* 启动模拟观察抛体运动。

1.3 实验结果

如果你正确实现了模拟过程和前向欧拉法，那么你应该能看到方块进行斜上抛运动。在运动过程中点击 *Stop* 按钮，就可以让所有物体暂停运动。在任何时候点击选中物体，它的速度会以蓝色箭头的形式表示。图 1.3 展示了初始化和运动过程中的状态。

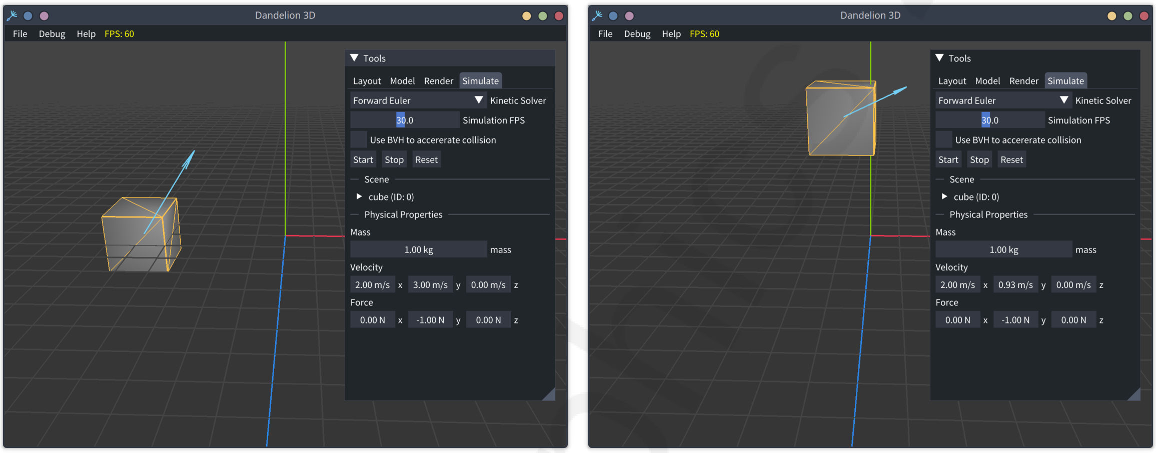

图 1.3: 设置方块的初始状态（左图）和方块运动过程中的状态（右图）

当你想要结束模拟时，只要点击 *Reset* 按钮就可以恢复到上一次点击 *Start* 时的状态。

1.4 提交和验收

本实验需要提交的截图有两张：设定好初始状态的截图和模拟过程中的截图，后者可以在任意时间截取，但必须能看到方块。其他内容按照标准要求即可。

验收时，你需要回答关于时间步长设置与前向欧拉法的问题，并现场以 Release 模式编译、演示你的程序。如果你能够正确地实现固定步长的模拟过程（即 `Scene::simulation_update` 和 `Object::update` 函数），可以获得 5 分；如果在此基础上进一步实现了前向欧拉法，那么你可以再获得 10 分，共计 15 分。

第 2 章 更好地求解运动方程

前向欧拉法的误差会随着迭代时间增长而愈发明显，在本实验（编号 2.13）中，你将实现三种更好的求解方法。

2.1 实验内容

了解一些衡量微分方程数值解法优劣的标准，并理解前向欧拉法的缺陷。在此基础上阅读资料了解隐式欧拉法、半隐式欧拉法和四阶龙格-库塔法的原理、实现这三种求解方法。

2.2 指导和要求

2.2.1 前向欧拉法的缺陷

如果你试着修改实验 2.12 中抛体运动的参数，将 *Simulation FPS* 改为 5 ，再用水平面网格作标尺来观察抛体运动，就可以发现模拟结果出现了明显的误差。例如设置初始 $\mathbf{x}_0 = (-4, 0, 0), \mathbf{v}_0 = (2, 2, 0), \mathbf{F} = (0, -1, 0)$ ，理论上当方块重新落回水平面时应该在 $(4, 0, 0)$ 处，实际上会偏离大约 0.2 格。

因为我们并没有准确地求解微分方程，所以出现误差完全是可以理解的，这是微分方程离散化求解的固有缺陷。将前向欧拉法转换为递推方程后求解 $\mathbf{x}(t)$ 的过程称为**显式时间积分**，它的不稳定性尤其明显，当时间步长较大时便容易暴露出来。

2.2.2 更好的时间积分方法

除了显式时间积分（前向欧拉法），研究者们还探讨了很多更稳定的微分方程数值解法，本实验中你需要学习并实现其中三种：

- 隐式欧拉法 (Implicit Euler Integration)，请自己查找资料学习。
- 半隐式欧拉法 (Semi-Implicit Euler Integration, or Symplectic Euler Integration)，参考 Wikipedia - Semi-implicit Euler method 实现。
- 四阶龙格-库塔法 (4-th Runge-Kutta Integration)，参考 Wikipedia - Runge-Kutta Methods 实现。

上面列出的是我们推荐的参考资料，你也可以自己寻找其他资料学习，大体是共通的。

2.2.3 要求

首先，一些求解方法不仅需要当前状态，还需要上一步状态，因此 `Object` 类中维护了一个 `prev_state` 属性。你需要修改之前实现的 `Object::update` 函数，在更新物体状态之前先更新上一步状态（将当前状态赋值给上一步状态、下一步状态赋值给当前状态）。

之后，你需要填写 *simulation/solver.cpp* 中的三个函数：

- `backward_euler_step`
- `symplectic_euler_step`
- `runge_kutta_step`

2.3 实验结果

如果你正确地完成了上述三种求解方法，那么模拟结果应该与前向欧拉法大致相同，但设置更大的时间步长（即更低的模拟帧率）时更加准确。

2.4 提交与验收

本实验需要提交的截图有三张，分别是用三种方法模拟同一运动过程的场景，截图时间不必相同。其他内容按照标准要求即可。

验收时，你需要现场回答关于这几种方法的问题，并以 Release 模式编译程序，演示任意物体的运动动画。实现隐式欧拉法可得 3 分、半隐式欧拉法可得 3 分、四阶龙格-库塔法可得 4 分，共计 10 分。

第 3 章 朴素的碰撞检测

实现了运动动画之后，我们还希望物体之间能够发生交互，碰撞就是一种常见的交互形式。在本课程的实验中，朴素的碰撞检测（实验 2.14）是指用简单遍历的方式检测 mesh 与 mesh 间碰撞的过程。

3.1 实验内容

了解碰撞检测的工作过程、理解射线-三角形求交的推导方法，借助动量定理推导两个物体碰撞后产生的速度变化，并实现碰撞检测和响应。

3.2 指导和要求

3.2.1 碰撞检测

在实现运动动画的过程中，我们是以离散的时间步为单位模拟运动的。碰撞检测正是基于运动模拟的结果来判断两个物体是否发生碰撞的，具体来说，碰撞检测的输入是“所有物体在 t 时刻的状态”，而不是某一段运动过程；我们要实现的碰撞检测算法，就是检查这一状态下物体之间是否重叠，存在重叠时认为发生了碰撞，因此碰撞检测的过程实质上是物体求交的过程。

笔记 这种只检查特定时间点状态的思路称为离散碰撞检测 (Discrete Collision Detection, DCD)，当某物体运动速度足够大时，就可以在一个时间步内穿过被碰撞的物体，从而产生“穿模”现象。相对地，精确地计算碰撞时间点从而避免穿模的方法称为连续碰撞检测 (Continuous Collision Detection, CCD)。有兴趣了解更多相关知识的同学可以学习 GAMES 103 课程 [2]。

在不同的应用领域，碰撞检测使用的求交对象也不同。例如很多游戏对碰撞检测的精度要求不高，所以经常使用包围盒作“碰撞体”，将物体包围盒相交判定为物体碰撞。在本实验中，我们直接用物体的 mesh 作为“碰撞体”，通过检测 mesh 相交来检测物体碰撞。这就涉及到 mesh 相交的定义问题，即我们认为什么样的两个 mesh 是相交的。

如果两个 mesh 都是封闭且可定向的，那么它们直观上来说就是一个“实心”物体的表面。此时，如果第一个 mesh 的内点集合 A 与第二个 mesh 的内点集合 B 交集非空，那么这两个 mesh 的“内部空间”就重叠，这是一种很直观的相交定义，在几何领域有不少应用。然而 mesh 的形状千变万化，这导致求两个内点集合的交集相当困难。

我们可以换一种定义方法：如果第一个 mesh 中某条边 e 与第二个 mesh 中某个面片 f 穿插，那么它们就相交。这种定义同时也给出了检测两个 mesh 是否相交的算法：遍历第一个 mesh 的每一条边，检查这条边是否与第二个 mesh 的任意一个面片相交；只要整个过程中发生了至少一次相交，两个 mesh 就相交。这个算法有两重循环，内层循环中只有一个简单的操作：边与面片求交。

在本实验中，你只需要处理三角形网格，因此需要考虑的情况只有边和三角形求交，即线段和三角形求交。如果用参数形式 $\mathbf{o} + t\mathbf{d}$ 表示射线，那么线段就是射线上 $t \in [a, b]$ 的部分。只要先将射线与三角形求交，再检查得到的 t 值是否属于相应的区间即可判断线段与三角形是否相交。

我们取边的一个顶点 $\mathbf{v}_1$ 作为射线起点，$\mathbf{v}_1$ 到 $\mathbf{v}_2$ 的方向作为射线方向，则这条边就是射线上 $t \in [0, \|\mathbf{v}_2 - \mathbf{v}_1\|]$ 的部分。关于射线求交的算法请参考 Whitted-Style Ray-Tracing（实验 2.4）的文档。

3.2.2 碰撞响应

我们已经定义了如何检测碰撞，但还没有定义碰撞后物体的运动状态会发生什么改变，即碰撞响应。在本实验中，碰撞响应包括位置和速度的变化。

当我们检测到碰撞时，两个物体各有一部分相互重叠。这是一种暂时的不稳定状态，如果不消除它，那么下一帧还会继续检测到碰撞，有可能导致发生碰撞的两个物体卡在一起。为此，检测到碰撞并计算完速度变化量后，需要将物体移回它在这个时间步开始时的位置（即传递给 `step` 函数的位置），让它在下一个时间步处于不重叠的状态。

在上一小节中，我们用枚举边、求边-三角形交点的方法检测碰撞。为了方便，接下来我们始终枚举物体 2 的边，用它们与物体 1 的面片求交，并称物体 2 为碰撞者，物体 1 为被碰撞者。这个碰撞关系如图 3.1 所示。

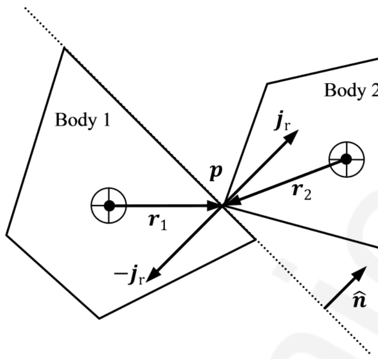

图 3.1: 两个物体碰撞的示意图 [1]

真实世界的碰撞过程比较复杂，物体的平动与转动两部分运动状态都可能改变。简单起见，我们在实验中只考虑物体的平动，并且认为发生的是完全弹性碰撞且碰撞时间极短。在碰撞前后，物体只有速度的变化，角速度始终是零。这样的过程遵循动量定理，根据两个物体的质量和碰撞前的速度，就可以求出它们碰撞后的速度。

参考资料 [1] 中给出了一般形式下碰撞响应的推导过程，在这里我们只推导不含转动的部分。如图 3.1，碰撞点（边与三角形的交点）为 $\mathbf{p}$ ，物体 2 在该点处的法向为 $\mathbf{n}$ 。我们认为两个物体给对方的冲量 $\mathbf{j}_r$ 和 $-\mathbf{j}_r$ 分别与 $\mathbf{n}$ 同向和反向，因此这个相对冲量可以写成 $\mathbf{j}_r = j_r\mathbf{n}$ 。根据牛顿第三定律和动量定理，可以得到二者碰撞后的速度为：

$$\begin{aligned} \mathbf{v}_1' &= \mathbf{v}_1 - \frac{j_r}{m_1}\mathbf{n} \\ \mathbf{v}_2' &= \mathbf{v}_2 + \frac{j_r}{m_2}\mathbf{n} \end{aligned} \tag{3.1}$$

而一次完全弹性碰撞前后，两个物体的相对速度 $\mathbf{v}_r = \mathbf{v}_2 - \mathbf{v}_1$ 满足如下关系：

$$\mathbf{v}_r' \cdot \mathbf{n} = -\mathbf{v}_r \cdot \mathbf{n} \tag{3.2}$$

将 3.1 式代入 3.2 式中，就可以求出冲量的模 j_r ，进而求出碰撞后的速度。

3.2.3 要求

你需要完成的任务有：

- 修改 *scene/object.cpp* 中的 `Object::update` 函数，填写其中完成碰撞检测、碰撞响应的部分。
- 填写 *utils/ray.cpp* 中的 `naive_intersection` 函数（请参考实验 2.4 的指导）
- 将 *CMakeLists.txt* 中 `src/utils/ray.cpp` 一行取消注释，并取消链接 `dandelion-ray` 这个静态库。

完成后，你可以从最简单的方块开始进行碰撞试验，启动方法与之前完全相同。如果你正确地实现了检测算法，看到的结果应该大致符合（平动）动量守恒。用 Debug 模式编译程序时，建议将所有模型的总面数控制在 50 面以下；用 Release 模式编译时，建议将总面数控制在 200 面以下，否则可能产生严重的卡顿。

3.3 提交与验收

你需要加载两个简单的物体并使之相互碰撞，并提交碰撞前与碰撞后的截图。要求在碰撞前、后分别选中两个物体以显示它们的速度，因此你总共需要提交四张截图。其他内容按照标准要求即可。

验收时，你需要现场以 Release 模式编译程序并测试两个简单物体碰撞的过程。如果你正确地实现了碰撞检测，可以得 5 分；在此基础上正确地实现碰撞响应可以再得 10 分，共计 15 分。实现碰撞检测的判断标准是碰撞后至少要发生变化，例如碰撞后两个物体变为静止可以得 5 分，而保持原先速度直接相互穿过则不得分。

第 4 章 用几何数据结构加速碰撞检测

上一个实验中实现的碰撞检测方法虽然大致正确，但运行开销太大。在加速碰撞检测（编号 2.15）这个实验中，你将学习使用 BVH 加速射线求交的过程，从而大大加快碰撞检测。

4.1 实验内容

了解用于划分空间的几何数据结构，掌握 BVH (Bounding Volume Hierarchy) 的用法，并实现用于加速射线求交的 BVH 数据结构。

4.2 指导和要求

由于实验 2.5 的文档中已经讲解过关于 BVH 的知识，我们不再重复说明，请直接参考实验 2.5 的指导部分。

4.2.1 要求

按照实验 2.5 的要求填写函数、修改项目，完成后重新进行碰撞试验。在进入物理模拟模式后，请勾选 *Use BVH to accelerate collision* 选项再开始模拟。

你需要加载 *cow.dae* 和 *cube.obj* 并测试让它们发生碰撞。

4.3 提交和验收

提交的截图与实验 2.14 类似，同样是四张，其他内容按照标准要求即可。

验收时，你需要现场以 Release 模式编译并运行程序，测试相对复杂的物体发生碰撞的过程。你还需要打开菜单栏上的 *Debug -> Debug Options -> Show BVH* 选项，展示构建好的 BVH 。如果你能正确地构建 BVH ，可以得 5 分；如果能正确地进行碰撞检测与响应，可以得 10 分，共计 15 分。

参考文献

[1] Colinvella. *Collision response rigid impulse reaction*. Wikimedia Commons. published under CC BY-SA 3.0 license. 2010. URL: https://en.wikipedia.org/wiki/File:Collision_response_rigid_impulse_reaction.png.

[2] 王华民. *GAMES103：基于物理的计算机动画入门*. 2021. URL: https://games-cn.org/games103/.